To detect cycle in an undirected graph using BFS we can follow below steps -

- Create an adjacency list if we don't already have one.
- Create a queue of pair. The pair will store both node and it's parent node which will be crucial in subsequent steps.
- Add a starting node with parent as -1 b default in our Queue.
- Create a visited array/set.
- Loop while queue is not empty & mark the poped element as visited.
- Now, for it's neighbors, we check if they are not visited we just add them to queue. In case they are visited then we check if that neighbor is parent which is allowed because we come from parent node. However, if neighbor is visited & not parent node then there is a cycle.

Note :- For non-connected graphs we should perform above steps for each component since any of them may have a cycle.

Time complexity - $O(N+2E)$ where N is number of nodes and E is number of edges. (we can add + $O(N)$ to indicate that we visit every node once even if it's visited, we don't call BFS on that case that why we are not multiplying this component).
Space complexity - $O(N)$

```java
class Solution {
    // Function to detect cycle in an undirected graph.
    public boolean isCycle(int V, ArrayList<ArrayList<Integer>> adj) {
        // Code here
        boolean[] vis = new boolean[V];
        for (int node = 0; node < V; node++) {
            if (!vis[node] && bfs(node, -1, adj, vis))
                return true;
        }

        return false;
    }

    private boolean bfs(int node, int parent, ArrayList<ArrayList<Integer>> adj, boolean[] vis) {
        vis[node] = true;

        Queue<Pair> q = new LinkedList<>();
        q.offer(new Pair(node, parent));

        while (!q.isEmpty()) {
            int curNode = q.peek().node;
            int curParent = q.poll().parent;

            for (int ng: adj.get(curNode)) {
                if (ng == curParent)
                    continue;
                if (vis[ng])
                    return true;
                vis[ng] = true;
                q.offer(new Pair(ng, curNode));
            }
        }

        return false;
    }

    private class Pair {
        int node;
        int parent;
        Pair (int node, int parent) {
            this.node = node;
            this.parent = parent;
        }
    }
}
```